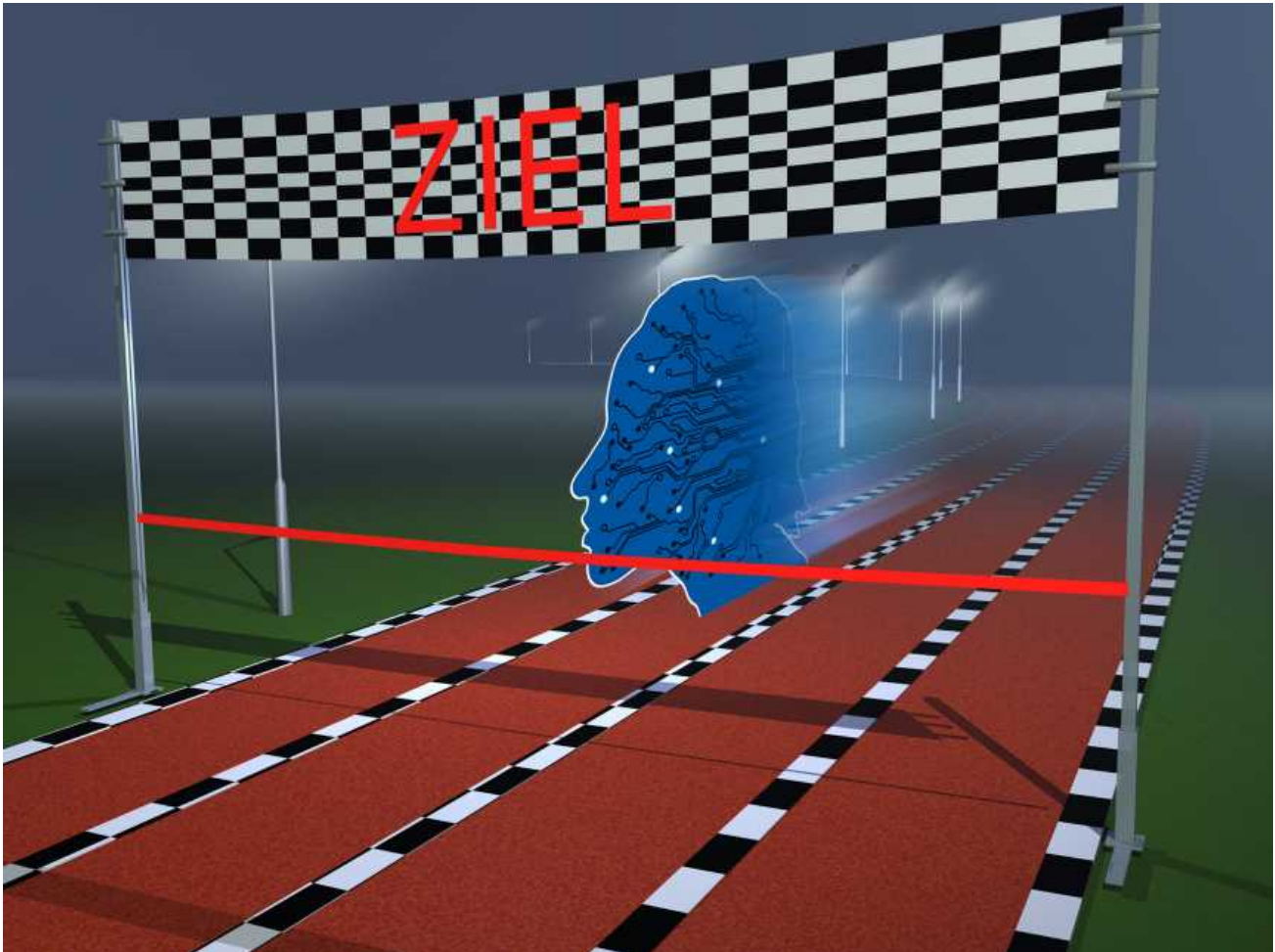


iX 8/2020 S. 134

PRAXIS

OBJEKTERKENNUNG



Deep-Learning-Tutorial: Einsatz des trainierten Modells auf der Zielhardware

Willkommen bei den Jetsons

Ramon Wartala

Der Weg trainierter Modelle auf die eigentlichen Zielsysteme fällt mitunter sehr verschieden aus. Dabei müssen Entwickler vor allem auf die Leistung und den Stromverbrauch achten.



- Rechenmodule der Jetson-Reihe stecken in Robotern und autonomen Fahrzeugen und können mit Hilfe energieeffizienter GPUs Objekte in Echtzeit erkennen.
- Die Jetson Development Kits sind auf einer Platine mit allen nötigen Peripherie- Anschlüssen vormontiert. Die größeren Systeme besitzen darüber hinaus einen PCIe-Anschluss, um SSD-Speichermedien anzuschließen.
- Das Open-Neural-Network-Exchange-Format(ONNX) soll einen anbieterübergreifenden Austausch von Modellen ohne erneute Konvertierung erlauben.

Die letzten beiden Teile dieses Tutorials haben viel über die Auswahl und die Beschaffung geeigneter Daten und ihre Aufbereitung für das Training von Deep-Learning-Modellen für die Objekterkennung erklärt. Dabei haben Entwicklerinnen anhand von Python und dem frei verfügbaren Google Colab gesehen, wie sich die Daten innerhalb einer Jupyter-Notebook-Umgebung nutzen lassen und wie sich die Trainingszeit durch Hardwarebeschleunigung verkleinern lässt.

Notebooks und Colab stehen hierbei stellvertretend für die Werkzeuge, mit denen man derzeit Machine-Learning-Modelle entwickelt und trainiert. Die Cloud eignet sich hervorragend als Trainingsplatz, was nicht nur daran liegt, dass alle großen Cloud-Anbieter Notebookumgebungen bereitstellen, sondern sich diese on demand skalieren lassen. Wenn absehbar ist, dass ein Modelltraining auf der gebuchten Hardware mehrere Tage in Anspruch nimmt, können Entwickler in der Cloud mit größerer Hardware die Trainingszeit verkürzen.

Bleibt die Frage, wie man trainierte Modelle auf die eigentlichen Zielsysteme bekommt. Dieser Schritt ist in vielen Anwendungsszenarien nötig, da nicht überall potente Grafikkarten aus Serverhardware zur Verfügung stehen. Heute finden sich die meisten Deep-Learning-Modelle bei mobilen Endgeräten, Industrierobotern oder PKWs. Diese Anwendungen laufen auf beschränkter oder zumindest spezialisierter Hardware. Besonders in mobilen Systemen ist der Stromverbrauch von entscheidender Bedeutung. Im Gegensatz zu GPU-Modulen in Servern, die schnell mehrere 100 Watt für den Betrieb benötigen, müssen mobile Geräte den Koeffizienten aus Leistung und

Dabei unterstützen diese spezialisierten Prozessoren die schnellere Ausführung der in Deep-Learning-Modellen verwendeten Mathematik. Chips wie Apples A12 Bionic oder Samsungs Neural Processing Units (NPU) beschleunigen die Operationen um ein Vielfaches bei niedrigem Stromverbrauch und entlasten so die CPU.

Seit 2014 bietet NVIDIA mit der Jetson-Familie spezialisierte Hardware an. Die stromsparenden Rechenmodule finden sich sowohl in Robotern als auch autonomen Fahrzeugen und ermöglichen Bilderkennung in Echtzeit mit Hilfe energieeffizienter GPUs. Den Anfang der Modellserie machte der Jetson TK1, der bereits die für Jetson-typischen Komponenten in sich vereinigte: ein Entwicklungsboard, eine Tegra genannte 64-Bit-ARM-CPU, eine NVIDIA-GPU, wie man sie aus handelsüblichen Grafikkarten kennt, Hauptspeicher, einen zum Raspberry PI kompatiblen, 40-poligen GPIO-Header und diverse andere Anschlüsse.

NVIDIA bietet die Rechenmodule in zwei Formfaktoren an: Die Development Kits richten sich an Entwickler und kommen auf einer Platine vormontiert mit allen nötigen Peripherieanschlüssen. Für den Einsatz im Embedded-Bereich sind die SoC Module mit PCIe-Anschluss gedacht.

Egal ob Samsung, Apple oder NVIDIAs Jetson – gemein ist diesen Systemen, dass man ein zuvor erzeugtes und trainiertes Deep-Learning-Modell an die Zielhardware anpassen muss. Bei NVIDIA dient dazu das SDK TensorRT, das auf CUDA und cuDNN basiert (siehe Abbildung 2). Das TensorRT-Framework erfüllt dabei mehrere Aufgaben.

Es stellt Parser für Caffe, TensorFlow und das ONNX-Format (Open Neural Network Exchange) zur Verfügung. Weiterhin enthält es einen Optimierer, der die Operationen des Deep-Learning-Modells auf die Zielhardware zuschneidet, und bietet eine Laufzeitumgebung, um die optimierten Modelle auszuführen.

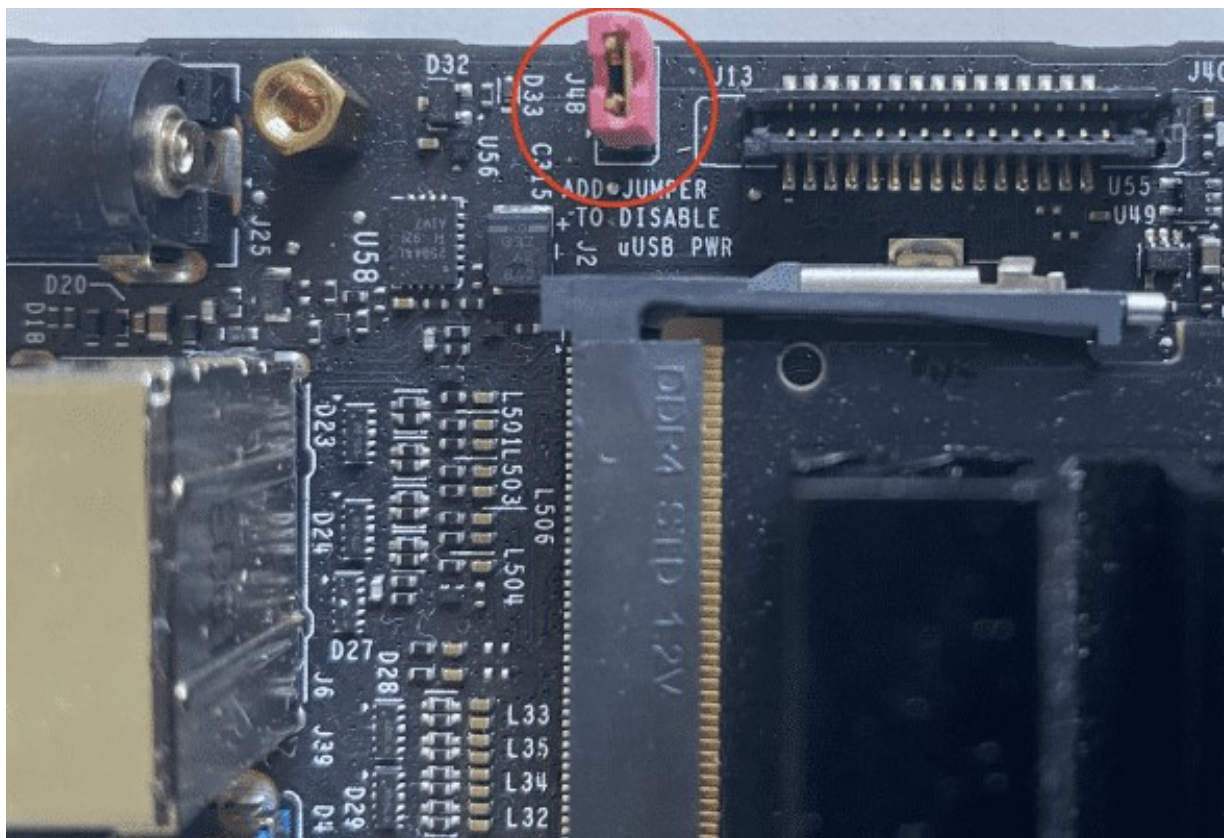
TensorRT lässt sich in C++ oder über die Python-API programmieren. Obwohl Python auf den Jetsons im Vergleich zu C++ deutlich langsamer ist, lassen sich dank TensorRT und dem hardware-nahen CUDA die eigentlichen Deep-Learning-Operationen ohne große Performanceeinbußen ausführen.

Was es braucht, um Jetsons einzurichten



Das Jetson Nano Developers Kit (DevKit) bringt nur wenig Zubehör mit. Für eigene Experimente lohnt es sich, ein paar Ergänzungen zu erwerben oder bereit zu legen. Um das System überhaupt booten zu können, braucht man eine MicroSD-Karte mit mindestens 32 GB oder besser 64 GB. Neben dem Startimage (alle Links unter ix.de/z92c) sollte noch genügend Platz für weitere Software sowie eigene Programme und Daten sein. Alternativ lassen sich dafür die USB-3.0-Eingänge nutzen.

Für die Installation zwingend ist darüber hinaus eine USB- oder Funktastatur, eine Maus oder ein Trackpad sowie ein passendes Netzteil. Dafür kommt entweder ein vorhandenes MicroUSB- oder Raspberry-Pi-Netzteil (5V~2A) in Frage. Beide liefern dem Nano rund 5W. Das reicht allerdings nicht aus, um die ganze Leistung des kleinen Rechenzwergs zu Tage zu fördern. Dazu muss man dem System volle 10W Leistung über ein entsprechendes Netzteil zuführen. Zusätzlich braucht es jedoch ebenfalls einen entsprechenden Jumper auf Pin J48 (Abbildung 1).



Die rosa Jumperbrücke auf Jumperpin 48 unterbindet den Einsatz eines USB-Netzteils beim Jetson Nano und ermöglicht die 10W-Nutzung des Modus 0 mit höheren Taktraten (Abb.1).

Wer etwas ambitioniertere Projekte plant, kann seit Mitte Mai mit dem Jetson Xavier NX Developers Kit eine leistungstärkere Variante erwerben. Eine Übersicht über die Ausstattungsunterschiede beider Systeme findet sich in der Tabelle „Überblick über

unterschiedlichen Leistungen der beiden Powermodi“. Weitere Informationen liefert der Kasten „Ungleiche Brüder – Nano vs. Xavier NX“ (mehr dazu in iX 6/2020, S.

34).

Überblick über die unterschiedlichen Leistungen der beiden Powermodi

Boardkomponenten	Jetson Nano 5W Modus	Jetson-Nano-10W-Modus (MaxN)	Jetson-Xavier-NX-10W-Modus	Jetson-Xavier-NX-15W-Modus
Leistungsaufnahme	5 Watt	10 Watt	10 Watt	15 Watt
Mode-ID	1	0	1	0
Genutzte CPU-Kerne	2	4	6	6
Maximaler CPU-Takt (MHz)	918	1479	2 mit 1500, 4 mit 1200	2 mit 1900, 4/6 mit 1400
Maximaler GPU-Takt (MHz)	640	921,6	800	1100

Gut zu sehen ist, dass im sogenannten MaxN-Modus doppelt so viele CPU-Kerne und ein höherer GPU-Takt zur Verfügung stehen. Wenn noch ein USB-Anschluss frei ist, lässt sich an ihm eine handelsübliche USB-Kamera anschließen. Alternativ können Entwickler zur Objekterkennung eine vorhandene CSI-Kamera nutzen, welche anschlussgleich zum Raspberry Pi ist.

TensorFlow und Keras

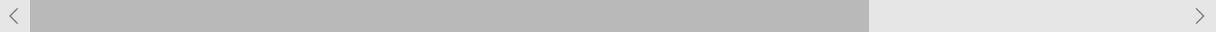
Seit August 2018 gibt es für Jetson eine TensorFlow-Implementierung. Mit TensorFlow 2 ist die Keras-API von François Chollet fester Bestandteil. Bislang fehlt TensorFlow in JetPack, der Framework-Sammlung für die Jetsons, und muss manuell nachinstalliert werden. Dies kann entweder direkt im Terminal des Jetson oder remote über eine SSH-Verbindung erfolgen. Zuerst gilt es, die nötigen Voraussetzungen in Form benötigter Softwarepakete zu installieren (Listing 1).

Listing 1: Softwarepakete aufspielen

```
apt-get update
apt-get install libhdf5-serial-dev hdf5-tools libhdf5-dev zlib1g-dev zip libjpeg-dev
apt-get install python3-pip
pip3 install -U pip testresources setuptools
pip3 install -U numpy==1.16.1 future==0.17.1 mock==3.0.5 h5py==2.9.0 keras==2.2.4
```



```
pip3 install --pre --extra-index-url https://developer.download.nvidia.com/ coi
```



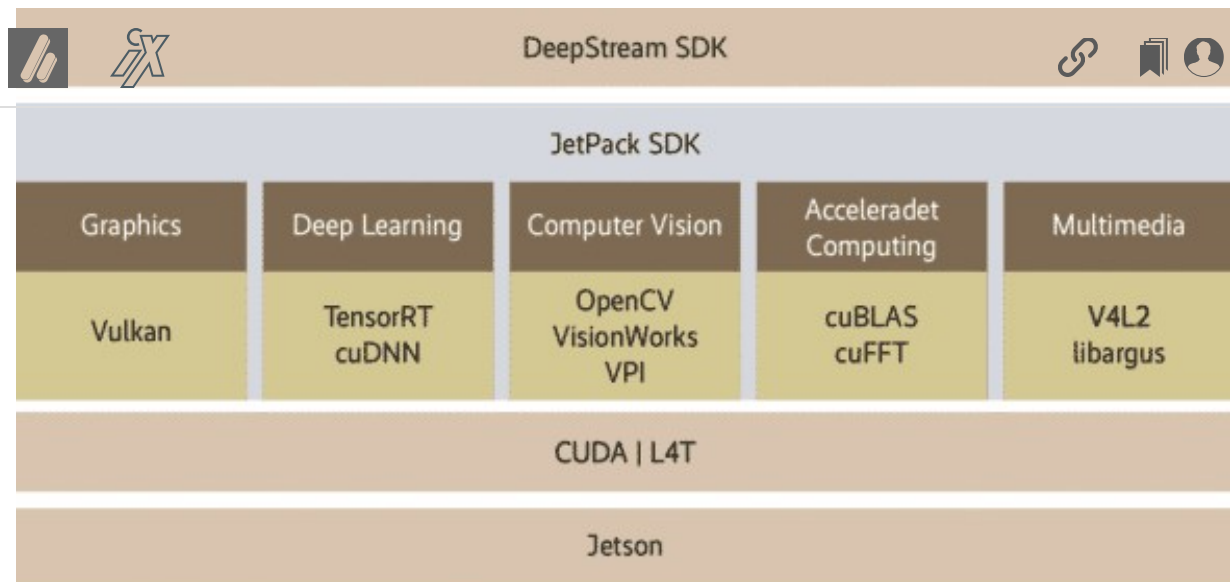
installieren. Damit kann man auf dem Jetson die gleiche Entwicklungsumgebung nutzen, wie sie in Teil 1 und 2 dieses Tutorials auf Googles Colaboration Plattform zum Einsatz kam. Entsprechend kann der Entwickler den Python-Quellcode für die Objekterkennung direkt auf dem Jetson ausführen. Der Befehl

```
git clone https://github.com/rawar/ ix-tut-yolov3.git
```

installiert den Quellcode des Tutorials und kopiert das trainierte Keras-Modell. Wer sich die Zeit vor Augen führt, die der Jetson Nano benötigt, um das Modell zu laden und auf das Beispielbild anzuwenden, wird sich einer Enttäuschung nicht ganz wehren können. Es dauert ganze 32 Sekunden, bis die Objekte auf einem Beispielbild erkannt wurden.

Warum dauert das so lange? Das YOLOv3-Modell sollte auf dem Jetson Nano nahezu in Echtzeit Objekte auf Bildern erkennen können. Die Erklärung für die lahme Erkennungsrate ist, dass das trainierte YOLOv3-Modell nicht optimal auf der Jetson Hardware läuft. TensorFlow erkennt die GPU zwar, aber die Hardwareoptimierung mit Hilfe von TensorRT läuft nicht einfach automatisch ab.

Abbildung 2 zeigt einen Überblick über die Softwarebibliotheken, die auf die CUDA-Architektur der darunterliegenden GPU ausgerichtet sind. Neben speziell beschleunigten Bibliotheken für die Berechnung von Matrizen und andere Aufgaben aus der linearen Algebra (cuBLAS) sowie zur Berechnung schnellerer Fourier-Transformationen (cuFFT) offeriert JetPack mit cuDNN und TensorRT zwei Bibliotheken für Deep-Learning-Anwendungen. Erst diese Abstraktionsschicht ermöglicht es Entwicklern, das selbsttrainierte YOLOv3-Modell performant auszuführen.



Basierend auf NVIDIAs CUDA-Bibliothek und Linux for Tegra (L4T) bietet JetPack eine Reihe optimierter Bibliotheken und Module für unterschiedliche Anwendungsfelder (Abb. 2).

Damit das Modell in den Genuss des versprochenen Geschwindigkeitsschubs kommt, muss man es im ersten Schritt in eine Form bringen, die TensorRT verarbeiten kann. TensorRT bietet Modellparser für Caffe, TensorFlows UFF und das ONNX-Format. Beispielsweise zeigt die Verwendung von Letzterem, wie sich das trainierte Keras-Modell in eine für TensorRT lesbare Form bringen lässt.

Universelle Modelle: Das ONNX-Format


Schon seit Jahren versuchen Firmen wie Facebook, Intel, Amazon, Microsoft, NVIDIA und viele andere, ein offenes Standardformat für den Framework-übergreifenden Austausch von Machine-Learning-Modellen zu etablieren. Das Open-Neural-Network-Exchange-Format (ONNX) ermöglicht einen Framework-übergreifenden Austausch trainierter Modelle. Zu den Hauptbestandteilen von ONNX zählen Definitionen für Berechnungsgraphen, Operatoren, Funktionen und Standard-Datentypen.

Für die unterschiedlichen Machine-Learning-Frameworks finden sich auf der GitHub-Seite von ONNX geeignete Konverter für TensorFlow und Keras, PyTorch, CoreML, Caffe und Matlab. Für die Konvertierung von Keras nach ONNX existiert zum Beispiel mit `keras2onnx` eine Python-Implementierung. Alles, was man braucht, um ein trainiertes Keras-Modell in das ONNX-Format zu übertragen, ist die nötige Software

```
pip install git+https://github.com/microsoft/onnxconverter-common
```

```
pip install git+https://github.com/onnx/keras-onnx
```



 der Python-Code aus Listing 2. Dazu baut die Entwicklerin das YOLOv3 Modell mit dem im zweiten Teil beschriebenen Code auf und liest die gespeicherten Gewichte.

Der keras2onnx-Konverter erledigt den Rest.

Listing 2: Von Keras nach ONNX mit keras2onnx

```
import os
os.environ['TF_KERAS'] = '1'
import onnx
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Add
from tensorflow.keras.models import Model
from core.yolov3 import YOLOv3, decode
import onnx
import keras2onnx

input_size = 416
input_layer = tf.keras.layers.Input([input_size, input_size, 3])
feature_maps = YOLOv3(input_layer)

bbox_tensors = []
for i, fm in enumerate(feature_maps):
    bbox_tensor = decode(fm, i)
    bbox_tensors.append(bbox_tensor)


keras_model = tf.keras.Model(input_layer, bbox_tensors)

keras_model.load_weights("./starwars_yolov3")
onnx_model = keras2onnx.convert_keras(keras_model, keras_model.name)
onnx.save_model(onnx_model, "starwars_yolov3.onnx")
```

Richtung Laufzeitumgebung

Der erste Schritt für eine beschleunigte Ausführung unseres Star-Wars-Figurenmodells ist damit gemeistert. Anschließend gilt es, das ONNX-Modell mit Hilfe des TensorRT-Parsers einzulesen, und eine so genannte Engine zu überführen, eine spezielle, hardware-optimierte Version des Modells. Dieses Vorgehen funktioniert dabei sowohl auf den Jetsons als auch auf anderen NVIDIA-Plattformen, die TensorRT akzeptieren.

Der Beispielcode findet sich im GitHub-Repository im Unterverzeichnis jetson und be-

reibt die Konvertierung des bestehenden Star-Wars-ONNX-Modells in eine Ten-
sorRT Laufzeitumgebung. Diese als Engine bezeichnete, CUDA-optimierte Version des
Modells lässt sich als Binärdatei abspeichern.

Das Builder-Objekt hilft der Entwicklerin, selbst Hand an die Modelloptimierung zu legen. Die Genauigkeit des Modells bestimmt die Performance und lässt sich über den verwendeten Datentyp ändern. Die Jetsons bieten mit ihrem INT8- und FP16-Modus zwei Möglichkeiten, auf etwas Vorhersagepräzision zu Gunsten der Ausführungsgeschwindigkeit zu verzichten. Konkret werden bei diesen Modi die trainierten Gewichte in reduzierte Datentypen übertragen, was die Modelle verkleinert und ihre Ausführungszeit verkürzt.

Übersicht über die unterschiedlichen Ausführungszeiten des trainierten Star-Wars-Modells bei unterschiedlichen Modellpräzisionen

Jetson-Modell	Genauigkeit	Modellgröße	Erkennungszeit pro Bild
Nano	FP32	573 MB	521 ms
Nano	FP16	288 MB	408 ms
Xavier NX	FP32	573 MB	625 ms
Xavier NX	FP16	122 MB	101 ms

Entwickler erreichen über

```
builder.fp16_mode = True
```

quasi aus dem Stand, nur durch Änderung der Modellpräzision, eine sechsfach höhere Verarbeitungsgeschwindigkeit.

Mit TensorFlow schlussfolgern

Die erzeugte TensorRT-Engine liegt nun auf dem Jetson für die Ausführung bereit. Jetzt beginnt der etwas aufwändigere Teil des Unternehmens Echtzeitobjekterkennung von Star-Wars-Figuren. Das Bild, auf dem die Figuren erkannt werden sollen, egal ob aus einer Datei, einem Video- oder Webcam-Stream, muss in das trainierte Modell gebracht und die resultierende Ausgabe am letzten Layer verarbeitet und interpretiert werden. Mit TensorFlow lässt sich das leicht umsetzen, da viele Hilfsfunktionen existieren, um die entsprechenden Eingangs- und Ausgangs-Tensoren zu konvertieren.

Man sich mit dem zugrundeliegenden Modell nicht gut auskennt, dem helfen Tools wie Netron, ein Python-Debugger, oder einfache print-Anweisungen, um die Dimensi-

onen der Eingabe- und Ausgabe-Tensoren zu verstehen. Wählt man diese sowohl für die zu verarbeitenden Bilder als auch für die zurückgelieferten Klassifizierungen falsch, kann schnell das berühmte GIGO (garbage in, garbage out) entstehen.

Listing 3 zeigt einen Teil des predict_image.py-Scripts. Die TensorRT-Engine wird geladen, das Bild im richtigen Format bereitgestellt und die Inference ausgeführt. Das YoloV3-Modell liefert drei Ausgaben für die unterschiedlichen Bildunterteilungen zurück. Als Eingabe braucht es einen Tensor der Form (416, 416, 3) für Breite und Höhe in Pixeln und einem RGB-Farbwert. TensorRT liefert als Ausgabe aus dem Modell drei einfache Arrays zurück. Anschließend müssen Entwickler sie in die richtigen Formate bringen. Im Falle von YOLOv3 sind es drei Tensoren mit den Dimensionen (1, 30, 13, 13), (1, 30, 26, 26) und (1, 30, 52, 52).

Listing 3: predict_image.py

```
...
TRT_LOGGER = trt.Logger(trt.Logger.ERROR)
trt_runtime = trt.Runtime(TRT_LOGGER)






print(f"Lade TensorRT Engine {engine_file_path}")
trt_engine = common.load_engine(trt_runtime, engine_file_path)

inputs, outputs, bindings, streams = common.allocate_buffers(trt_engine)
context = trt_engine.create_execution_context()

inputs[0].host = image_processed
print(f"Starte die Objekterkennung auf Bild {input_image_path}")
inference_start_time = time.time()

# Liefert die drei YOLOv3 Ausgaben zurück
trt_outputs = common.do_inference_v2(
    context,
    bindings=bindings,
    inputs=inputs,
    outputs=outputs,
    stream=streams
)

print(f"TensorRT Erkennungszeit: {} ms".format(int(round((time.time() - inference_start_time) * 1000))))
```

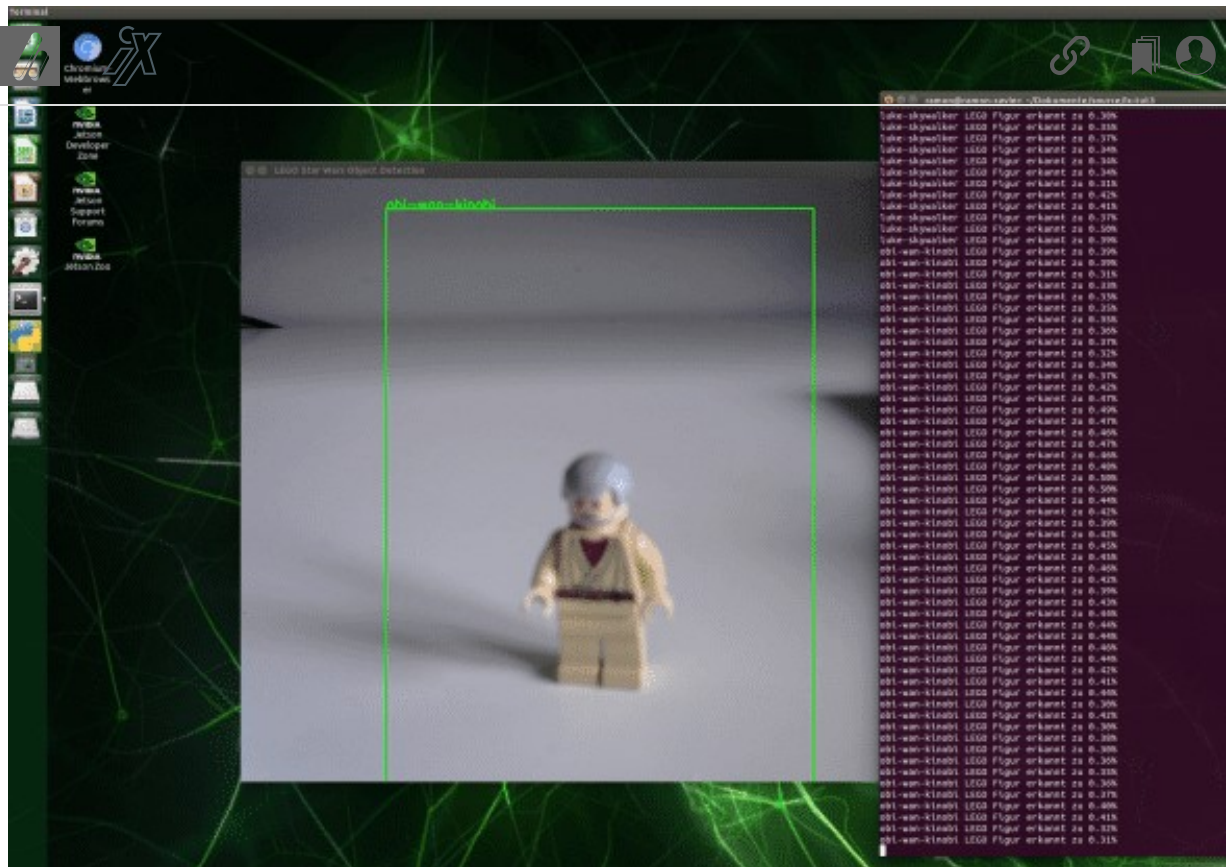


```
trt_output_shapes = [(1, -1, new_height // 32, new_width // 32),  
                     (1, -1, new_height // 16, new_width // 16),  
                     (1, -1, new_height // 8, new_width // 8)]  
  
trt_outputs = [output.reshape(shape) for output, shape in zip(trt_outputs, ou  
...  
< >
```

Die darin enthaltenen X- und Y-Koordinaten, die Objektklassen und ihre Wahrscheinlichkeiten muss man in einem mehrstufigen Prozess extrahieren. Dazu bringt die Entwicklerin die drei Tensoren in die Formen (507, 10), (2028, 10) und (8112, 10) und verbindet sie. Die Funktion `nms` innerhalb von `predict_image.py` entscheidet, welche der erkannten Objektboxen ausgewählt werden.

Achtung, Kamera läuft!

Eine der interessantesten Anwendungen für die Jetsons ist die Echtzeiterkennung von Objekten in Video- oder Kamerastreams. Mit Hilfe von OpenCV und einer angeschlossenen CSI- oder USB-Webcam lässt sich basierend auf dem bereits existierenden Quellcode eine einfache Anwendung erstellen. Listing 4 enthält einen Teil des Quellcodes für die Bildverarbeitung mit OpenCV. Hierbei stammt das Bild nicht aus dem Dateisystem, sondern von einer angeschlossenen USB-Kamera. Das von der Kamera gelesene Bild wird über die gleiche Inference-Pipeline wie zuvor verarbeitet (Abbildung 3).



Mit Hilfe von OpenCV und einer USB-Kamera erkennt der Jetson die Star-Wars-Figur. (Abb. 3)

Dank TensorRT- und FP16-Modelloptimierung liegt die Erkennungsrate auf dem Xavier NX jetzt etwa bei zehn Bildern pro Sekunde. Das ist noch nicht Echtzeit, aber für den reinen Python-Ansatz schon eine ordentliche Steigerungsrate zu einem Bild alle 32 Sekunden, das man mit normalem, GPU-optimierten tf.keras erzielt hatte.

Listing 4: Mit OpenCV-Code Objekterkennungsprogramme implementieren

```
import cv2
import numpy as np
import argparse
import os
import time
from PIL import Image
import tensorrt as trt
import img_utils
import common
...
cap = cv2.VideoCapture(0)
```





```
if not cap.isOpened():
    print("Kann die Webcam, nicht öffnen")
    exit()

while cap.isOpened():

    status, frame = cap.read()


    if not status:
        print("Kann kein Bild laden")
        exit()

    image_resized = cv2.resize(frame, (new_width, new_height), interpolation=cv2.INTER_LINEAR)
    image_resized = np.array(image_resized, dtype=np.float32, order='C')
    image_resized /= 255.0
    image_processed = np.transpose(image_resized, [2, 0, 1])
    image_processed = np.expand_dims(image_processed, axis=0)
    image_processed = np.array(image_processed, dtype=np.float32, order='C')

    inputs[0].host = image_processed

    trt_outputs = common.do_inference_v2(
        context,
        bindings=bindings,
        inputs=inputs,
        outputs=outputs,
        stream=streams
    )


    trt_outputs = [output.reshape(shape) for output, shape in zip(trt_outputs, output_shapes)]
    ...
```



Im Geschwindigkeitsrausch

Um die Ausführung von dem hier gezeigten YOLOv3 und andere Deep-Learning-Modelle weiter zu beschleunigen, stehen verschiedene Möglichkeiten zur Verfügung. Man könnte das Modell durch INT8-Gewichte noch kompakter machen. Statt des ausgewachsenen YOLOv3-Modells mit seinen über 800 Schichten und über 61 Millionen Gewichten lässt sich sein kleinerer Bruder, der YOLOv3-tiny, nutzen. Es ist nur rund ein Sechstel so groß und man kann es dadurch noch schneller ausführen.



 Jetsons besitzen verschiedene Leistungsmodi, um verschiedene Workloads abzu-
bilden. Mit Hilfe des Kommandozeilen-Tools `nvpmodel` lässt sich dieser Leistungsmodi-

us konfigurieren. Um den Jetson Nano vom stromsparenden Modus 1 (5W) in den Modus 0 (10W) zu bringen, kommt folgender Befehl zum Einsatz:

```
nvpmodel -m 0
```

Auch bei Xavier NX ermöglicht der Aufruf des Modus 0 die höchste Stufe der Leistungsfähigkeit bei gleichzeitig höchster Leistungsaufnahme. Besonders bei rechenintensiven Modellen sollte immer der leistungsfähigere Modus aktiviert sein. Zusätzlich lässt sich über spezielle Frameworks für mehr Geschwindigkeit beim Verarbeiten von Video- und Kamerastreams sorgen. NVIDIAs DeepStream etwa ist besonders für die Echtzeitanalyse von Videostreams konzipiert.

Weitere Verbesserungen

Ein Tutorial wie dieses muss den Spagat zwischen Nachvollziehbarkeit und Alltagstauglichkeit schaffen. Das gelingt einer gehosteten Jupyter-Notebookumgebung besser als einer lokalen Installation auf dem heimischen PC oder Mac. Für ein echtes Produktivsystem ist das hier Gezeigte zu umständlich.

Seit JetPack 4.2.1 gibt es für die Jetsons eine eigene Container Runtime (NCR), die es ermöglicht, Docker einzubinden. Die Idee dabei: Für das Training aufwändiger Deep-Learning-Modelle verwendet man den gleichen Container auf einem leistungsstarken Server. Dieser wandert dann, ohne Änderungen, für die Ausführung direkt auf das Edge-Device. NVIDIA bietet nicht nur die Container-Runtime, sondern auch einige Container an.

Möchte man einen ganzen Fuhrpark von Edge-Computern mit den neuesten Modellen aus dem letzten Training ausstatten, muss man sich weitere Gedanken machen. So bietet zum Beispiel Amazon SageMaker mit Neo und AWS IoT Greengrass eine Deployment-Plattform für etliche Endgeräte, darunter auch die Jetsons.

Ungleiche Brüder – Nano vs. Xavier NX

NVIDIAs Xavier NX ist das designierte Upgrade für jene, die bereits an die Grenzen des Jetson Nano gekommen sind und einfach mehr von allem für ihre Anwendungen benötigen. Mehr Hauptspeicher für größere Modelle, mehr Platz für d

Interferenz verschiedener Modelle, mehr Kameras und mehr gleichzeitige Video-streams kodieren und dekodieren. Abbildung 4 zeigt einen Vergleich beider Geräte.

So liefert der Xavier NX genug Leistung, um gleichzeitig Objekte über die Kamera und Sprache über ein Mikrofon am USB-Port zu erkennen. Dies überforderte den Nano schlicht.



Links der Jetson Nano mit dem passiven Kühlkörper und rechts der neue Jetson Xavier NX mit einem aktiv geregelten Lüfter (Abb. 4).

Jetson Nano vs. Jetson Xavier NX		
Technische Daten	Jetson Nano	Jetson Xavier NX
Erscheinungsjahr	Mitte 2019	Mitte 2020
Grafikprozessor	Maxwell mit 128 Cores	Volta mit 384 Cores und 48 Tensor-Cores
NVIDIA Deep Learning Accelerator Engines	-	2
CPU	64-Bit ARM CPU mit 4 Kernen	64-Bit ARM CPU mit 6 Kernen
Arbeitsspeicher	4GB LPDDR4x RAM, 25.6 GB/s	8 GB LPDDR4x RAM, 51,2 GB/s



Jetson Nano vs. Jetson Xavier NX



Datenspeicher	microSD	microSD + NVMe
Videokodierung	1x 4K mit 30 Bildern/s, 4x 1080p mit 30 Bildern/s, 9x 720p bei 30 Bildern/s (H.264/H.265)	2x 464 MP/Sek, 2x 4K bei 30 Bildern/s (HEVC), 6x 1080p bei 60 Bilder/s (HEVC)
Videodekodierung	1x 4K mit 60 Bildern/s, 2x 4K mit 30 Bildern/s, 8x 1080p mit 30 Bildern/s, 18x 720p bei 30 Bildern/s (H.264/H.265)	2x 690 MP/Sek., 2x 4K bei 60 (HEVC), 12x 1080p bei 60 (HEVC), 32x 1080p bei 30 (HEVC)
Kamera	Bis zu 2 Kameras	Bis zu 6 Kameras
Anschlüsse	Gigabit-Ethernet, M.2 Key E, GPIO, I2C, I2S, SPI, UART	Gigabit-Ethernet, M.2 Key E, GPIO, I2C, I2S, SPI, UART
Bildschirm	HDMI und DisplayPort	2x Multi-Mode DisplayPort 1-4 und HDMI 2.0
USB	4x USB 3.0, USB 2.0 Micro-USB	4x USB 3.0, USB 2.0 Micro-USB
Leistungsaufnahme	5W oder 10W	10W oder 15W
Abmessungen	100 mm x 80 mm x 29 mm	100 mm x 80 mm x 29 mm
Preis	~100 Euro	~400 Euro

Fazit

Dieses dreiteilige Tutorial hat grundlegende Techniken zur Erstellung eigener Objekterkennungssysteme mit Hilfe von Python und den gängigen Deep-Learning-Bibliotheken Tensorflow und Keras demonstriert. Der letzte Abschnitt thematisierte, wie sich ein einmal trainiertes Modell auf eine andere Zielplattform mit völlig anderen CPU und GPU bringen lässt. Daneben lag der Fokus auf Optimierung. Doch auch eine dreiteilige Artikelserie kann nur an der Oberfläche kratzen. Moderne Deep-Learning-Deployment-Pipelines setzen wie moderne Backend-Systeme auf containerisierte Microservices, die an dieser Stelle nicht beleuchtet werden konnten. Auch sind die Anhänger anderer Deep-Learning-Frameworks wie PyTorch, theano und Caffe2 etwas zu kurz gekommen und die Einschränkung auf Python hat dem ein oder anderen C/C++-Entwickler sicherlich nicht immer gefallen.



Ein wichtigen Punkt sollte man bei all der Spielerei mit Lego und Star Wars nicht außer Acht lassen: Die Verfügbarkeit immer günstigerer Systeme wie den Jetsons mit ei-

ner immer leistungsfähigeren Objekterkennung, etwa hinsichtlich des Datenschutzes und der Überwachung, könnten Schattenseiten haben, die es trotz des Spaßes am Gerät zu beachten gilt. (csc@ix.de)

Quellen

Alle Listings und Links unter <https://www.ix.de/z92c>

Ramon Wartala

ist Director Data Science bei SinnerSchrader (Part of Accenture Interactive) in Hamburg und berät mit seinem Team Kunden in Datenarchitekturen für Machine-Learning-Anwendungen.

■ Kommentieren



Leserbrief schreiben



Auf Facebook teilen



Auf Twitter teilen

Kontakt

Impressum

Datenschutzhinweis

Nutzungsbedingungen

Mediadaten



